COMPUTER SCIENCE TECHNICAL REPORT

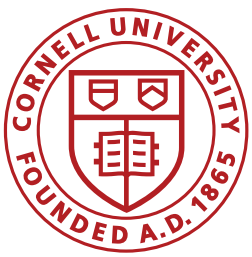
CSTR 2016-04 04/19/2016

Bolt: Uncovering and Reducing the Security Vulnerabilities of Shared Clouds

Christina DELIMITROU
CORNELL UNIVERSITY

Christos KOZYRAKIS
STANFORD UNIVERSITY

April 19, 2016



Bolt: Uncovering and Reducing the Security Vulnerabilities of Shared Clouds

Abstract

Cloud providers routinely schedule multiple applications per physical host to increase cost efficiency. The resulting interference in shared resources leads to performance degradation and, more importantly, security vulnerabilities. Interference can leak important information ranging from the placement of a service to confidential data, like private keys.

We present Bolt, a practical system that accurately detects the type and characteristics of applications sharing a cloud platform, based on the interference an adversary sees on shared resources. Bolt leverages practical data mining techniques for detection that operate online and require 2-5 seconds. In a 40-server shared cluster, Bolt correctly detects 81% out of 108 diverse batch and interactive workloads. Extracting this information enables a wide spectrum of previously-impractical cloud attacks, including denial of service (DoS), resource freeing (RFA) and co-residency attacks. For example, Bolt can successfully launch difficult to detect, host-based DoS attacks, with only a fraction of the resources and time needed by a conventional distributed DoS that cause the tail latency of the victim to increase by up to 140x. Finally, we show that, while advanced isolation techniques, such as cache partitioning, lower detection accuracy, they are insufficient to eliminate these vulnerabilities. To do so, one must either disallow core sharing, or only allow it between threads of the same application, leading to significant inefficiencies and performance penalties respectively.

1. Introduction

Cloud computing has reached proliferation by offering *resource flexibility* and *cost efficiency* [1–3]. Cost efficiency is achieved through multi-tenancy, i.e., by co-scheduling multiple VMs or containers on the same physical host to increase utilization. However, multi-tenancy leads to interference in shared resources, such as last level caches, hard drives, or network switches causing performance unpredictability [4–6]. More importantly, unmanaged contention also leads to security and privacy vulnerabilities [7]. This has prompted significant work on side-channel attacks [8, 9], distributed denial of service attacks (DDoS) [9–11], data leakage exploitations [8, 12], and attacks that pin-point target VMs in a cloud system [13–16]. Most of these schemes leverage the lack of strictly enforced resource isolation between co-scheduled instances and the naming conventions cloud frameworks use for machines to extract confidential information from victim applications, such as encryption keys.

This work presents *Bolt*, a practical system that can extract detailed information about the type and characteristics of applications sharing resources in a cloud system. Bolt uses online data mining techniques to quickly determine the amount of pressure an application puts in each of several shared resources. We show that this information is sufficient to determine the framework type (e.g., Hadoop), functionality (e.g., SVM), and dataset characteristics of a co-scheduled application, as well as the resources it is most sensitive to. Bolt periodically collects statistics on the resource pressure applications incur and projects this signal against datasets from previously-seen workloads. Since detection repeats periodically, Bolt accounts for changes in application behavior and can distinguish between multiple co-residents on the same physical host. We validate Bolt's detection accuracy with a controlled experiment in a 40-server cluster using virtualized instances. Out of 108 co-scheduled victim applications, including batch and real-time analytics, and latency-critical services like key-value stores and databases, Bolt identifies 81% of workloads correctly. We also used Bolt to determine the most commonly-run applications on Amazon EC2 and the extent to which applications are colocated. We find a very small number of application classes that consistently account for most of the compute cycles across time intervals, days and availability zones.

Obtaining this information enables Bolt to make several cloud attacks *practical* and *difficult to detect*. For example, we use Bolt to launch host-based DoS attacks that use the information regarding the victim's resource sensitivity to inject carefully-crafted contentious programs that degrade the victim's performance. In the 40-server cluster, Bolt's DoS attack translates to a tail latency increase of up to `140×` for interactive workloads. Unlike traditional DoS attacks that saturate compute and memory resources, Bolt maintains low utilization, by only introducing interference in the most critical resources, making it resilient to common DoS mitigation techniques, such as load-triggered live VM migration. It also does not require external, costly load generators aimed at the victim service, that take time to impact performance, and would trigger DoS detection mechanisms. We have also used Bolt to launch resource freeing attacks (RFA) that additionally force the victim to yield its resources to the adversary [17], and VM co-residency detection attacks that pinpoint where in a large shared cluster a specific application resides before negatively impacting its performance [18]. In all cases we show that the information obtained through data mining is critical to escalate the impact of the attack, reduce the time and cost it requires, and to make it difficult to detect.

Finally, we examine to what degree current isolation techniques can alleviate these security vulnerabilities. We analyze baremetal systems, containers, and virtual machines with techniques like thread pinning, memory bandwidth isolation, and network and cache partitioning. These are the main isolation techniques available today, and while they progressively re-

duce application detection accuracy from 81% to 50%, they are not sufficient to eliminate these vulnerabilities. We show that the only method currently available to reduce accuracy down to 14% is core isolation, where an application is only allowed to share cores with itself to avoid malicious co-residents. However, since application threads contend for shared on-chip resources performance degrades by 34% on average. Alternatively, if we allocate more cores per application to mitigate performance unpredictability, we end up with resource underutilization and a big reduction in cloud efficiency. We hope that this study will motivate public cloud providers to introduce stricter isolation solutions in their platforms and systems architects to develop fine-grain isolation techniques that provide strong isolation and performance predictability at high utilization.

2. Related Work

Performance unpredictability is a well-researched problem in public clouds that stems from network congestion, platform heterogeneity, resource interference, software bottlenecks and temporal load variation [19–21, 21–26]. A lot of recent work has proposed isolation techniques to reduce unpredictability by eliminating resource interference [27–31]. With Bolt, we show that unpredictability due to interference also hides security vulnerabilities, since it enables ad adversary to extract information about an application's type and resource characteristics. Below we discuss related work with respect to cloud vulnerabilities, such as VM placement detection, DDoS, and side-channel attacks.

VM co-residency detection: Cloud multi-tenancy has motivated a line of work on locating a target VM in a public cloud. Ristenpart et al. [13] showed that the IP machine naming conventions of cloud providers allowed adversarial users to narrow down where a victim VM resided in a large-scale cluster. Xu et al. [32] and Herzberg et al. [16] extended this study, resulting, in part, in cloud providers changing their naming conventions, reducing the effectiveness of network topology-based co-residency attacks. Following this evolution Varadarajan et al. [18] evaluated the susceptibility of three cloud providers to VM placement attacks, and showed that techniques like virtual private clouds (VPC) render some of them ineffective. Xu et al. [15] studied the extent of co-residency threats in EC2 and the efficiency of their detection using network route traces. Bates et al. [33] proposed a system where adversarial VMs introduce traffic congestion in host NICs, which is then detected by remote clients. Similarly, Zhang et al. [14] designed HomeAlone, a system that detects VM placement by issuing side-channels in the L2 cache during periods of low traffic. Finally, Han et al. [34] proposed VM placement strategies that defend against placement attacks, although they are not specifically geared towards public clouds. With Bolt, we show that leveraging simple data mining techniques on the pressure applications introduce in shared resources increases the accuracy of VM co-residency detection significantly. Bolt does not rely on knowing the cloud's network topology or host IPs, making it resilient against recent techniques, such as VPCs.

Performance attacks: Once an adversary locates a target victim application, it can negatively affect its performance. Distributed Denial of Service attacks [17, 35–37] in the cloud have increased in number and impact over the past years. This has generated a lot of interest in detection and prevention techniques [38, 39]. Gupta et al. [10] outline the characteristics of cloud facilities that make DDoS attacks likely, and propose profiling VMs to detect network DDoS attacks. Bakshi et al. [11] develop a system that detects abnormally high network traffic that could signal an upcoming DDoS attack, while Crosby et al. [40] and Edge [41] propose a new DoS attack relying on algorithmic complexity that drives CPU usage up. Finally, apart from DoS, resource-freeing attacks (RFAs) also hurt the performance of a victim, while additionally forcing it to yield its resources to the adversary [17]. While such systems degrade the victim's performance, they require significant compute and network resources, and they are prone to defenses such as live VM migration that cloud providers are introducing to mitigate unpredictability from resource saturation. In contrast, Bolt launches host-based DoS attacks on the same machine as the victim that leverage the information on the resources the application is sensitive to, and keep resource utilization moderate, therefore not triggering defense mechanisms.

Side-channel attacks: There are also attacks that attempt to extract confidential information from co-scheduled applications, such as private keys [42–45]. Zhang et al. [12] propose a system that launches side-channel attacks in a virtualized environment, and cross-tenant side-channel attacks in PaaS clouds [46]. The system overcomes three main challenges to extract an ElGamal decryption key: the frequent re-scheduling of VMs by the hypervisor, the noise in shared resources and the implications of core migrations. Wu et al. [47] on the other hand attempt to affect the performance of a victim application, by launching a covert-channel attack via the memory bus of a modern x86 processor. On the defense side, Perez-Botero et al. [48] analyze the vulnerabilities of common hypervisors, and Wang et al. [9] propose a system that specifically targets intrusion detection in cloud settings, while Liu et al. [8] and Varadajaran et al. [49] design scheduler-based defenses against covert- and side-channel attacks in resources such as the memory bus. The latter system controls the overlapping execution of different VMs and injects noise on the memory bus to prevent an adversary from extracting confidential information. Bolt does not rely on accurate microarchitectural event measurements through performance counters to detect application placement, and is therefore not affected by similar defenses that limit the fidelity of time-keeping and performance monitoring to thwart information leakage in side-channel attacks [50].

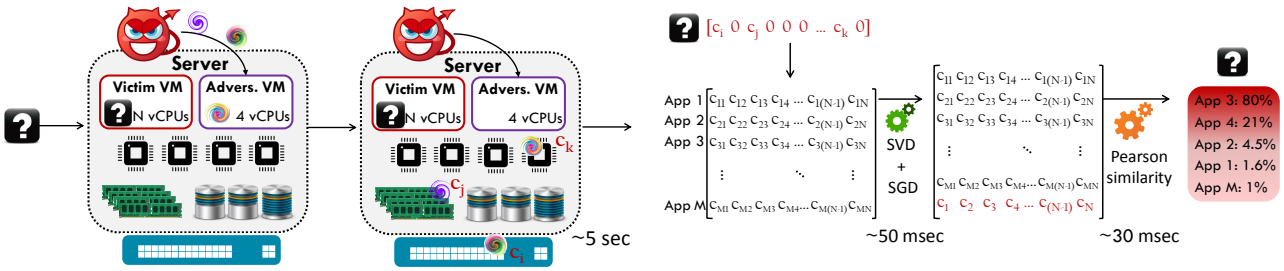


Figure 1: Overview of the application detection process. Bolt first measures the pressure co-residents place in shared resources, and then uses data mining to determine the type and characteristics of co-scheduled applications.

3. Bolt

3.1. Threat Model

Bolt targets IaaS providers that operate public clouds for mutually untrusting users. Multiple VMs can be co-scheduled on the same server. Each VM has no control over where it is placed, and no a priori information on other VMs on the same physical host. For now, we assume that the cloud provider is neutral with respect to detection by adversarial VMs, i.e., it does not assist such attacks or employ additional resource isolation techniques than what is available by default to hinder them. In Section 6 we explore how additional isolation techniques affect Bolt's detection accuracy.

Adversarial VM: An adversarial VM has the goal of determining the nature and characteristics of any applications co-scheduled on the same physical host, and negatively impacting their performance. Adversarial VMs start with zero knowledge of co-scheduled workloads.

Friendly VM: This is a normal VM scheduled on a physical host that runs one or more applications. Friendly VMs do not attempt to determine the existence and characteristics of other co-scheduled VMs. They also do not employ any schemes to prevent detection, such as memory pattern obfuscation [51].

3.2. Application Detection

Detection relies on inferring workload characteristics from the contention the adversary experiences in shared resources. For simplicity, we assume one co-scheduled victim job for now and generalize in Section 3.3. Figure 1 shows an overview of the system's operation.

Bolt instantiates an adversarial VM on the same physical host as the victim, friendly VM. Bolt uses its VM to run a few microbenchmarks of tunable intensity that put progressively more pressure each on a specific shared resource. The microbenchmarks are drawn from a set that can stress on-chip resources, such as functional units, and the different levels of the cache hierarchy, and off-chip resources, such as the memory, network and storage subsystems [52]. Each microbenchmark progressively increases its intensity until it detects pressure from the co-scheduled workload, i.e., until the microbenchmark's performance is worse than its expected value when running in isolation. The intensity of the benchmark at that point captures the pressure the co-scheduled application puts in shared resource i and is denoted c_i, where $i \in [1,N]$ and $N = 10$ and $c_i \in [0,100]$. [1]Large values of c_i imply high pressure in resource i. For unconstrained resources, e.g., last level cache, 100% pressure means that the benchmark takes over the entire resource capacity. For resources constrained through partitioning mechanisms, e.g., memory capacity in VMs, 100% corresponds to taking over the entire memory partition allocated to the VM.

Bolt uses 2-3 microbenchmarks for profiling, requiring approximately 2-5 seconds in total. Bolt randomly selects one core and one uncore benchmark to get a more rounded depiction of the co-resident's resource profile. If the core is not shared between the adversarial and friendly VMs, Bolt measures zero pressure in the shared core resource. In this case, it uses a third microbenchmark for an additional uncore resource (shared caches, memory, network or storage).

From this sparse signal we want to determine the application's pressure in all other resources, its type and characteristics. Online data mining techniques have recently been shown to solve similar problems in datacenter management in a fast and accurate way by finding similarities between unknown applications and previously-seen workloads [53]. The Quasar cluster manager, for example, proposed the use of collaborative filtering to find similarities in terms of heterogeneity, interference sensitivity, and provisioning. While collaborative filtering is useful in identifying resource usage similarities, it has no application-specific information, such as critical features, and is therefore insufficient to label the victim workloads and classify their resource profiles.

Practical data mining: Bolt instead feeds the profiling signal to a *hybrid* recommender system using feature augmentation [54, 55] that determines the application type and resource characteristics of victim workloads. The recommender combines *collaborative filtering* and *content-based similarity detection* [54, 56]. The former has good scaling properties, relaxed sparsity constraints and offers conceptual insight on similarities, while the latter exploits contextual information for highly accurate resource profile matching.

[1]The 10 resources are: L1 instruction and L1 data cache, L2 and last level cache, memory capacity and memory bandwidth, CPU, network bandwidth, disk capacity and disk bandwidth.

First, a collaborative filtering system recovers the pressure the victim application places in non-profiled resources [5, 57]. The system relies on matrix factorization with singular value decomposition (SVD) [58] and PQ-reconstruction with stochastic gradient descent (SGD) [59, 60] to find similarities between the new victim application and previously-seen workloads. SVD produces three matrices, U, Σ and V. The singular values σ_i in Σ correspond to similarity concepts, such as the intensity of compute operations or the correlation between high network and disk traffic. Similarity concepts are ordered by decreasing magnitude, with large singular values revealing stronger similarity concepts (higher confidence in workload correlation), while smaller values correspond to weaker similarity concepts and are typically discarded during the dimensionality reduction by SVD. Matrix $U_{(m,r)}$ captures the correlation between each application and similarity concept, and $V_{(n,r)}$ the correlation between each resource and similarity concept.

Once we have identified critical similarity concepts and discarded inconsequential information, we employ a content-based system that uses *weighted Pearson correlation* to compute the similarity between the resource profile u_j of a new application j and the applications the system has previously seen. Weights correspond to the values of the r more critical similarity concepts [2], and resource profiles to the rows of the matrix of left singular vectors U. Using traditional Pearson correlation would discard the application-specific information that certain resources are more critical for a given workload. Similarity between applications A and B is given by:

$$Weighted\ Pearson(A,B) = \frac{cov(u_A, u_B; \sigma)}{\sqrt{cov(u_A, u_A; \sigma) \cdot cov(u_B, u_B; \sigma)}} \quad (1)$$

where u_A the correlation of application A with each similarity concept σ_i, $cov(u_A, u_B; \sigma) = \frac{\sum_i \sigma_i (u_{Ai} - m(u_A;\sigma))(u_{Bi} - m(u_B;\sigma))}{\sum_i \sigma_i}$ the covariance of A and B under weights σ, and $m(u_A;\sigma) = \frac{\sum_i \sigma_i \cdot u_{Ai}}{\sum_i \sigma_i}$ the weighted mean for A. The output of the hybrid recommender is a distribution of similarity scores of how closely a victim resembles different previously-seen applications. For example, a victim may be 65% similar to a memcached workload, 18% similar to a Spark job running PageRank, 10% similar to a Hadoop job running an SVM classifier, and 3% to a Hadoop job running k-means. The end-to-end latency of the recommender is 80msec on average. Apart from application labels, this analysis yields information on the resources the victim is sensitive to, enabling several practical performance attacks (Section 5).

System insights from data mining: Before dimensionality reduction, each similarity concept corresponds to a shared resource. Different resources convey different amounts of information about a workload, and thus have different value to Bolt in terms of detection. The value of each similarity concept reflects how strongly it captures application similarities. For the controlled experiment of Section 3.4 that involves batch and latency-critical workloads, the resources with most value are the L1 i/d and last level caches, followed by compute intensity and memory bandwidth. While the exact resource order depends on the application mix, correlating similarity concepts to resources shows that certain resources are more prone to leaking information about a workload, and their isolation should be prioritized.

[2] We keep the r largest singular values, such that we preserve 90% of the total energy: $\sum_{i=1}^{r} \sigma_i^2 = 90\% \sum_{i=1}^{n} \sigma_i^2$.

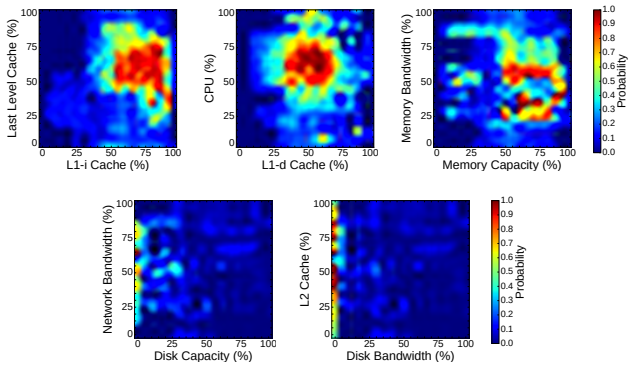


Figure 2: We show the correlation between the pressure in various resources and the probability of a co-scheduled application being memcached. For example, workloads with high LLC pressure and very high L1-i pressure have a high probability of being memcached.

Finally, we show how resource pressure correlates with the probability that an application is of a specific type. Figure 2 show the probability that an unknown workload is a memcached instance servicing mostly read requests with KB-range values, as a function of its measured resource pressure (details on methodology in Section 3.4). We decouple the 10 resources in 2D plots for clarity. From the heatmaps it becomes clear that cache activity is a very strong indicator of workload type, with applications with very high L1-i pressure and high LLC pressure corresponding to memcached with a high probability. Disk traffic also conveys a lot of information, with zero disk usage signaling a memcached workload with high likelihood. Similar graphs can be created to fingerprint other application types. The high heat areas around the red regions of each graph correspond to memcached workloads with different read:write ratios and value sizes, and other memory-bound workloads like Spark.

3.3. Challenges

Multiple co-residents: When a single application shares a host with Bolt, detection is straightforward. However, cloud operators colocate VMs more aggressively to maximize the infrastructure's utility. Disentangling the contention caused by several applications in shared resources is challenging. By default Bolt uses two benchmarks - one core and one uncore - to measure resource pressure. If the recommender system cannot determine the type of co-scheduled workload based on

Applications	Detection accuracy (%)	
	Least Load scheduler	Quasar scheduler
Aggregate	81%	83%
memcached	76%	78%
Hadoop	85%	89%
Spark	82%	83%
Cassandra	86%	88%
speccpu2006	81%	82%

Table 1: Bolt's detection accuracy in the controlled experiment with the least load scheduler and Quasar.

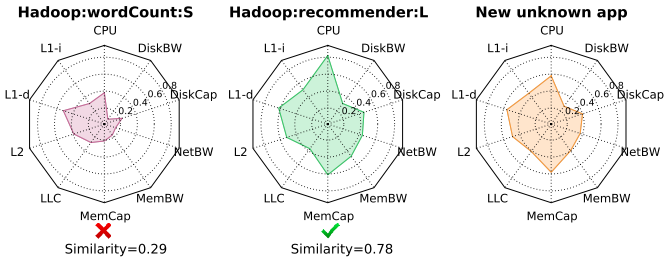


Figure 3: Star charts showing the resource profiles of two Hadoop and one unknown application. The closer the shared area is to a vertex the higher the resource pressure in the specific resource.

them, either the application type has not been seen before, or the resource pressure is the result of multiple co-residents. If at least one of the co-residents shares a core with Bolt, i.e., the core benchmark returns non-zero pressure, we profile with an additional core benchmark and use them to determine the type of co-scheduled workload. Because hyperthreads are not shared between multiple *active* instances, this allows accurately measuring pressure in core resources. The remainder of the uncore interference is used to determine the other co-scheduled workloads. This assumes a linear relationship in resource pressure for memory, network and storage bandwidth between workloads, which may introduce some inaccuracies (see Section 3.5).

Finally, there are cases where none of the co-scheduled applications shares a core with Bolt. In that case, the system must rely solely on uncore resource pressure to detect applications. To gain some insight on the variance of resource characteristics across co-residents, Bolt employs a *shutter profiling mode*, which involves frequent, brief profiling phases (100-200msec each) in uncore resources. The goal of this mode is to capture at least one of the co-scheduled workloads during a low-load phase, which would reveal the resource usage of only one of the co-residents. If a significant drop in resource pressure is measured in uncore resources, these resources drive the recommender system, following each profiling step. This technique is particularly effective for user-interactive applications that go through intermittent phases of low load; it is less effective for services with constant steady-state load, such as long-running analytics, or logging services. We will consider whether additional input signals, such as per-job cache miss rate curves, can improve detection accuracy for the latter workloads.

Application changes: Datacenter applications are notorious for going through multiple phases during their execution. Online services in particular follow diurnal patterns with high load during the day and low load in the night [1, 61, 62]. In addition, an application may not be immediately detected, especially during its initialization phase, or cloud users may purchase instances to run an application, and then maintain the resources to execute other applications, e.g., different analytics, or analytics over different datasets. In these cases, the output of Bolt's detection may become obsolete. We address this by periodically repeating the profiling and detection steps for co-scheduled workloads. Each iteration takes 2-5 seconds for profiling and a few milliseconds for the recommender's analysis. Section 3.4 shows a sensitivity study on how frequently detection needs to happen.

3.4. Detection Accuracy

We first evaluate Bolt's detection accuracy in a controlled environment, where all applications are known. We use a 40-machine cluster with 8 core (2-way hyperthreaded) Xeon-class servers, and schedule a total of 108 workloads, including batch analytics in Hadoop [63] and Spark [64] and latency-critical services, such as webservers, memcached [65] and Cassandra [66]. For each application type, there are several different workloads, with respect to algorithms, framework versions, datasets and input load patterns. Friendly applications are scheduled using a least-loaded (LL) scheduler that allocates resources on the machines with the most available compute, memory and storage resources. All workloads are provisioned for peak requirements to reduce interference. Even so, interference between co-residents exists, since the scheduler does not account for the sensitivity applications have to contention. In the end of this section, we evaluate how a scheduler that accounts for cross-application interference affects detection accuracy. The training set consists of 120 diverse applications including webservers, different analytics algorithms and datasets and different versions of key-value stores and databases. Increasing the training set size further did not improve the detection accuracy.

Training vs. testing set: All algorithms, datasets, and input loads used by applications in the training set are different from the ones used for testing workloads.

In each server we instantiate an adversarial VM running Ubuntu 14.04. By default the VM has 4vCPUs (2 physical cores) to generate enough contention (Figure 8 shows a sensitivity analysis on the VM size). The remainder of each machine is allocated to at least one or more friendly VMs. The maximum number of VMs (five in our setting) depends on the server configurations available. Friendly VMs run on either Ubuntu 14.04 or Debian 8.0. Adversarial VMs have no a priori information on the number and type of their co-residents. Applications are allowed to share a physical core, but must be on different hyperthreads (vCPUs). While sharing a single

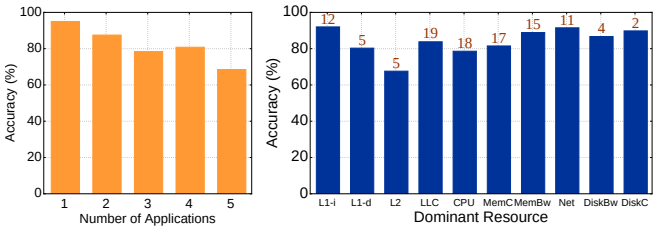


Figure 4: Detection accuracy as a function of the number of co-scheduled apps (left) and the applications' dominant resources (right).

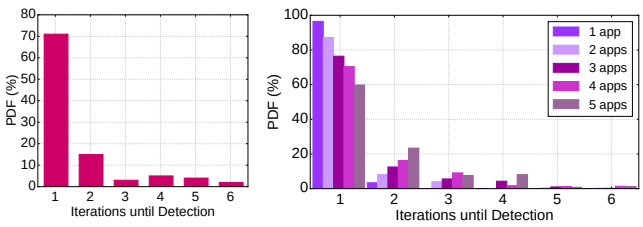


Figure 5: PDFs of the iterations required for correct detection in aggregate (left), and as a function of the number of co-residents (right).

hyperthread is common in private deployments hosting batch jobs in small containers [67], it is uncommon in public clouds, where 1 vCPU is the minimum guaranteed size of dedicated resources for non-idling instances.

Table 1 shows Bolt's detection accuracy, per application class, and aggregate. We signal a detection as correct if Bolt identifies correctly the framework (e.g., Hadoop) or service (e.g., memcached) the application uses, and the algorithm it performs, e.g., SVM classifier on Hadoop. We do not currently examine more detailed characteristics, such as the distribution of individual query types. Bolt correctly identifies the majority of jobs, 81%, and for certain application types like databases and analytics, the accuracy exceeds 85%. Misclassified applications are typically identified as workloads with the same or similar critical resources.

Per-application profiles: Note that although in Table 1 we group applications by programming framework or online service, each framework *does not correspond to a single resource profile*. Profiles of applications within a framework can vary greatly depending on functionality, complexity, and dataset features. Bolt's recommender system matches resource profiles to specific algorithmic classes and dataset features within each framework. Figure 3 shows an example where two Hadoop jobs, one running word count on a small dataset, and one running a recommender system on a very large dataset exhibit very different resource profiles. While the third application is also a Hadoop job it is identified as very similar to the recommender as opposed to word count.

Number of co-residents: The number of victim applications affects detection accuracy. Figure 4a shows accuracy as a function of the number of victims per machine. When the number of co-residents is less than 2, accuracy exceeds 95%. As the number increases accuracy drops, since it becomes progressively more difficult to differentiate between the contention caused by each workload. When there are 5 co-scheduled applications, accuracy is 67%. Interestingly, accuracy is higher for 4 than for 3 applications, since with 4 workloads the probability of sharing a core, and thus getting an accurate measurement of core pressure is higher. While aggressive multi-tenancy affects accuracy, large numbers of co-residents in public clouds are unlikely in practice [68].

Dominant resource: We now examine the correlation between the resource an application puts the most pressure on, and Bolt's ability to correctly detect it (Figure 4). The numbers on each bar show how many applications have each resource as dominant. In general, applications that are most easily detected are ones with high instruction cache (L1-i), memory bandwidth, network bandwidth and disk capacity pressure. These are workloads that fall in one of the following categories: latency-critical services with large codebases, such as webservers, in-memory analytics like Spark, and disk-bound analytics like Hadoop. Interestingly, L2 activity is a poor indicator of application type, in contrast to L1 and LLC, since it does not capture a significant change in the working set size (from 32KB to 256KB for our platforms).

Number of iterations: In this controlled experiment, we stop the detection process upon correct identification of a workload. For several applications this requires more than one iterations of profiling and data mining. Figure 5a shows the fraction of workloads that were correctly-identified after N iterations. 71% of victims only require a single iteration, while an additional 15% required a second iteration. A small fraction of applications needed more than two iterations, while applications that are not identified correctly until the sixth iteration did not benefit from additional iterations. The number of applications per machine also affects the iterations required for detection (Figure 5b). When a single application is scheduled on a machine, one iteration is sufficient for almost all workloads. As the number of victims per machine increases additional iterations are needed to disentangle the contention signals of each co-scheduled job.

Figure 6 shows a case of application detection over several iterations. The victim is a 4vCPU instance executing different consecutive applications. Detection occurs every 20 sec by default (see Figure 8 for a sensitivity study on the frequency of profiling). After the initial iteration, Bolt detects the application as `mcf` from SPECCPU2006. In the subsequent two iterations the interference profile of the victim continues to fit the characteristics of `mcf`. At time $t = 60$sec, Bolt detects that the application profile has changed and now resembles an SVM classifier running on Mahout (a machine learning library over Hadoop [63]). Similarly at $t = 180$sec the application changes again to a Spark data mining workload. Changes are typically captured within a few seconds.

Resource pressure: Figure 7 shows the correlation between the pressure of a victim in a given resource and Bolt's ability

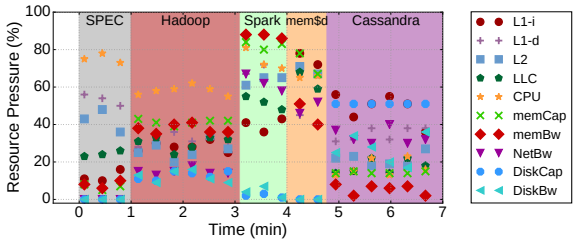


Figure 6: Example of workload phase detection by Bolt.

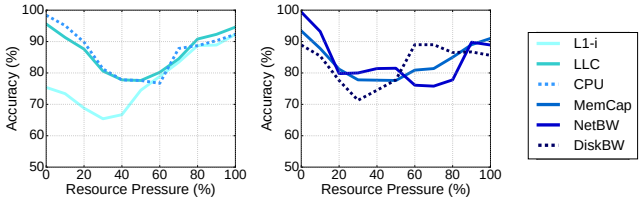


Figure 7: Detection accuracy as a function of the pressure victims place in various shared resources.

to correctly detect it. We plot detection accuracy for three core and three uncore resources; the results are similar for the remaining four. In almost all cases, very low or very high pressure carry the most value for detection. For disk bandwidth, accuracy remains high except for the 20-50% region which contains many application classes with moderate disk activity, including analytics in Hadoop, databases and graph applications. Resource pressure also affects the number of iterations Bolt needs to correctly identify a workload. For resources with a lot of value for detection, such as the L1-i and last level caches, when pressure is moderate several iterations are necessary for accuracy to increase. In contrast, for off-chip resources like network bandwidth this effect is less pronounced. In general, when moderate pressure hurts detection accuracy a lot (Figure 7), more iterations are needed for correct identification.

Scheduler: So far we have used a least loaded scheduler, which is commonly-used in datacenters [69, 70]. This scheduler does not account for resource contention between applications, leading to suboptimal performance [5, 71–74]. Recent work has shown that if interference is accounted for at the scheduler level both performance and utilization improve [4, 6, 53]. We now evaluate the impact of such a scheduler on Bolt's detection accuracy. Quasar [53] leverages machine learning to quickly determine which applications can be co-scheduled on the same machine without destructive interference. We use Quasar to schedule all victim applications and then inject Bolt in each physical host for detection.

Table 1 shows Bolt's accuracy with the least loaded scheduler (LL) and Quasar. Interestingly, accuracy increases slightly with Quasar, 2% on average, but in general the impact of the scheduler is small. There are two reasons for the increase; first, both LL and Quasar do not share a single hyperthread between applications, so core pressure measurements remain accurate. Second, because Quasar only co-schedules applications with different critical resources, it provides Bolt with a less "noisy" interference signal for uncore resources, making distinguishing between victim co-residents easier. Therefore reducing interference in software alone is not sufficient to mitigate such security threats.

Sensitivity analysis: Finally, we examine how design decisions in Bolt affect detection. Figure 8a shows how accuracy changes as profiling frequency decreases. For profiling intervals beyond 30 sec accuracy drops rapidly. If profiling only occurs every 5 minutes almost half the victims are incorrectly identified. Note that if workloads are long-running and mostly-stable, profiling can be less frequent without such an impact on accuracy. Figure 8b shows accuracy as a function of the adversarial VM's size. We examine sizes offered as on-demand instances by EC2 [2]. If the adversary has fewer than 4 vCPUs, its resources are insufficient to create enough contention to capture the co-residents' pressure. Accuracy continues to grow for larger sizes, however, the larger the adversarial instance is the less likely it will be co-scheduled with other VMs, nullifying the value of Bolt. Finally, Figure 8c shows accuracy as a function of the number of microbenchmarks used for profiling. A single benchmark is not sufficient to fingerprint the characteristics of a workload, however, using more than 3 benchmarks has diminishing returns in accuracy. Unless otherwise specified, we use 20sec profiling intervals, 4 vCPU adversarial VMs and 2 benchmarks for initial profiling.

3.5. Limitations

While Bolt can accurately detect most frequently-run workloads, it has certain limitations. First, when no application shares a core with the adversary, the system assumes linear relation between the co-residents' resource pressure in uncore resources. While this is true in some cases, it may not be generally accurate. We will explore alternative ways of distinguishing multiple co-residents in future work. Second, the system cannot differentiate between multiple applications running in a single instance and multiple instances running one application each. While this does not affect detection, if an side/covert-channel attack is aimed against a particular user, this information would increase the leverage of the adversary. Similarly, Bolt cannot currently distinguish between two copies of the same application sharing a platform at low load and one copy of the application that runs at higher load. Finally, Bolt relies on measuring resource interference. When isolation techniques that eliminate interference are in place detection accuracy drops (Section 6).

4. EC2 Cartography

We now use Bolt to find the most commonly-run applications on Amazon EC2 and their resource characteristics.

Disclaimer: Since we do not control externally-submitted applications, we cannot validate detection accuracy for this study. We rely on the validation of Section 3.4 where Bolt

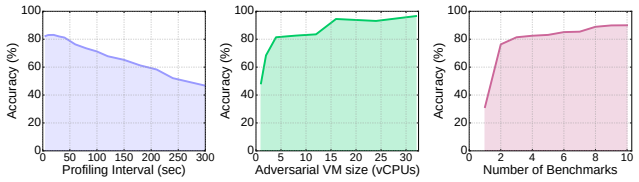


Figure 8: Sensitivity to: (a) profiling frequency, (b) adversarial VM size, and (c) profiling benchmarks.

correctly identifies similar applications. Nevertheless, there are reasons that may cause inaccuracies, discussed in 4.3.

Ethical considerations: To abide with the Amazon user agreement, we only conduct experiments that allow us to detect the types of applications running on EC2, but we do not exploit side- or covert-channel attacks against co-resident workloads. The performance implications of Bolt to victim applications are evaluated exclusively in a controlled local environment, and described in Section 5.

Detection setting: We request 430 4 vCPU on-demand instances, and verify that they are not placed on the same physical machine [18, 49].[3] The experiment is performed on a weekday between 11am and 2pm to record a period of high load. Resources are obtained from the `us-west-2` (Oregon) region. We use Bolt to determine the existence and profiles of co-scheduled applications. In **41%** of obtained instances the adversary was the *only* VM occupying the server, despite only requiring 4 vCPUs. [4] We continue to obtain instances until we have 430 adversarial VMs with at least one co-resident (in total **683** instances were examined).

We repeat the same setting during the night on a weekday, during the day of the weekend and during the day of a weekday in a different availability zone (`us-east-1`). To obtain 430 instances with at least one co-resident instance we had to request **810**, **785**, and **613** instances respectively. The training set is the same as in Section 3.4.

4.1. Application Types

Figure 9a shows the distribution (PDF) of detected application types. We group applications by framework class or service platform, e.g., Hadoop or SQL server. Because neighboring frameworks, such as BashReduce or Disco can have similar resource profiles to Hadoop, their applications may be misclassified as Hadoop workloads. To avoid overestimating the number of Hadoop jobs running on EC2, we label all such workloads as *Hadoop-like*. Note that these jobs experience the two-phase execution control flow of Hadoop, and exhibit strong similarities in resource pressure.

Interestingly a very small number of application classes dominates most of the activity in the examined cluster. As expected, these primarily include analytics, webservers and databases. These findings are well aligned with the services EC2 is starting to offer as SaaS [75]. Apart from the main five application classes, there is a long tail of less common workloads, which were encountered less than 20 times across all instances; including an instance running several SPECCPU2006 benchmarks. We also include a grey bar with applications that were not identified with satisfactory confidence (similarity over 75%). [5]These are applications that have either not been seen before by Bolt, or have conflicting idiosyncrasies in their behavior that do not allow them to be classified with high confidence in any of the known application types. Such "user" or "item" occurrences are known as *grey* and *black sheep* respectively in the recommender system community. They also include unknown workloads we detected multiple times each, but cannot label. A more diverse training set could assist with such applications.

Figures 9(b-d) show similar PDFs for the other three experiments. The top five classes remain consistent across time intervals, days and regions, although their order changes. For example, jobs on SQL servers are more common during the day than in the night, although their absolute difference is small. More interestingly, the number of application classes decreases significantly compared to the default scenario, both for the night and weekend experiments. Similarly the number of unknown applications drops, indicating that during periods of low load most applications are easily-identified and long-running. The difference between Zone 1 and Zone 2 is marginal, with slightly more classes in Zone 2. In general, we observe that a large fraction of cloud activity concentrates on a small number of frameworks which are independent of timing and the cloud's location.

	Fraction of servers (%)			
	1 app	**2 apps**	**3 apps**	**4 apps**
Weekday:Day:Zone1	61%	22%	16%	1%
Weekday:Night:Zone1	51%	23%	22%	4%
Weekend:Day:Zone1	72%	19%	7%	2%
Weekday:Day:Zone2	56%	25%	16%	3%

Table 2: Breakdown of the number of co-scheduled applications detected by Bolt in the EC2 setting.

4.2. Resource Usage

Cloud utilization: We now examine the degree of multi-tenancy on EC2. Table 2 shows the number of detected co-residents across the 430 instances for each experiment. The majority of hosts have 1-2 co-residents, while a small number of machines have 3 or more co-scheduled workloads. We did not find (or detect) servers with more than 4 co-residents. Co-scheduling is higher during periods of low load (night and weekend), indicating that VMs are consolidated to fewer machines to improve their utilization. However, even during periods of low load the number of co-scheduled applications

[3]We use several user accounts to launch the adversarial VMs, to reduce the probability of adversarial-adversarial VM co-residency.

[4]It is possible that there are dormant instances on the same machine that were not servicing an application at the time.

[5]From the validation study if similarity is over 75% the detection is correct with a very high probability.

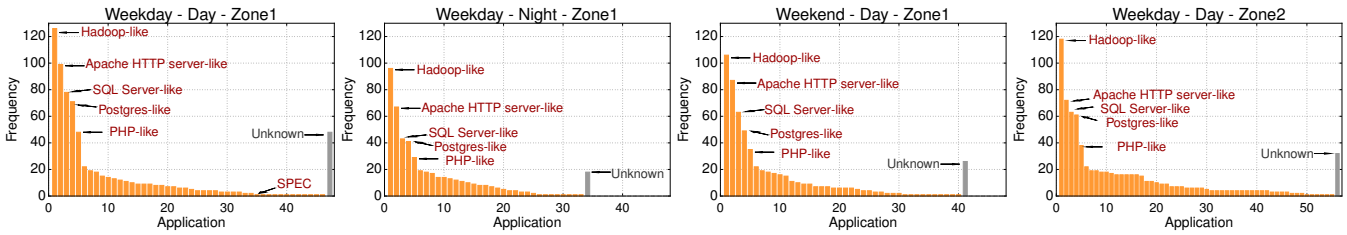


Figure 9: Application types detected on Amazon EC2 for different time intervals, days, and availability zones.

is not large, confirming studies that report very low utilization on virtualized clouds like EC2 [68].

Resource characteristics: We observe that for more than 50% of instances, the resource pressure in L1 i/d caches, L2 cache and CPU is significantly lower when measured during profiling than when inferred by Bolt. This concurs with the findings of Table 2 that co-scheduling is limited, with co-residents mostly sharing the memory, network and storage subsystems, but not specific cores. In contrast, inferred pressure for memory, network and storage bandwidth is very close to measure pressure. Since memory and disk capacity allocations are enforced in cloud instances, those measurements return zero pressure. In general, the resources that experience the highest pressure are the L1 i/d and last level caches, followed by compute and memory bandwidth. The remaining resources rarely come close to saturation, in part due to the limited amount of co-scheduling [68]. High resource pressure can be detrimental to performance, highlighting the need for practical and efficient isolation mechanisms that eliminate interference. In Section 6 we evaluate existing isolation mechanisms, and underline trade-offs between security and performance (or utilization).

4.3. Cartography Caveats

There are three main effects that may cause applications to be misclassified. First, neighboring frameworks, such as BashReduce and Hadoop can have similar resource patterns, causing the former to be misclassified as Hadoop workloads. We group all such applications under *Hadoop-like* to avoid overestimating workloads from any particular framework. Second, while our training set is sufficiently diverse for the validation study of Section 3.4, public clouds host a wider application spectrum. While Bolt can still determine the resource characteristics of these applications, it cannot assign a label to them, conservatively adding them to the unknown category. Third, aggressive multi-tenancy affects Bolt's detection accuracy. Beyond 5 co-residents detection accuracy drops to about 60%. In practice, cloud providers do not co-schedule non-idling instances this aggressively to prevent performance unpredictability [68]. We adopt the following methodology to further minimize misclassifications. We use a training set that includes many popular frameworks and online services, with a good coverage of resource characteristics. Additionally, we only consider an application as "detected" if confidence in detection is very high, above 75% for this experiment, beyond

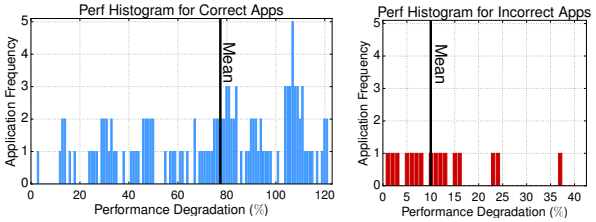


Figure 10: Performance degradation for correctly- and incorrectly-identified applications.

which point detection is almost always correct. Finally, while the precise application labels may be prone to such inaccuracies, determining the resource characteristics of a co-resident is sufficient to launch a wealth of performance attacks, as discussed in Section 5.

5. Security Attacks

Bolt makes several attacks in cloud settings practical and difficult to detect. Below we discuss three possible attacks that leverage the information obtained through detection.

5.1. Internal Denial of Service Attack

Attack setting: Denial of service attacks hurt the performance of a victim service by overloading its resources [11, 35, 39]. Specifically, in cloud settings they can be categorized in two types; *external* and *internal* (or host-based) attacks. External attacks are the most conventional form of DoS [35, 40, 41]. These attacks utilize many external servers to direct excessive traffic to the online services, flooding their resources, and hurting their availability. External DoS attacks affect mostly PaaS and SaaS systems, and include IP spoofing, synchronization (SYN) flooding, smurf, buffer overflow, land, and teardrop attacks.

On the other hand, internal DoS attacks affect a victim's performance by taking advantage of IaaS and PaaS cloud multi-tenancy to launch adversarial programs on the same host as the victim [13, 37, 39, 76]. For example, Ristenpart et al. [13] show how an adversarial user can leverage the IP naming conventions of an IaaS cloud to locate a victim VM and degrade its performance. Cloud providers are starting to build defenses against such attacks. For example, Amazon EC2, offers autoscaling that will scale out the number of instances of a service under heavy resource usage. This means that DoS attacks that simply overload a physical host are weak when such defenses are in place. Similarly, there is related

work on mechanisms that detect saturation in resources such as memory bandwidth and trigger VM migration [77]. We focus on host-based DoS attacks that are resilient against such defenses. Specifically, Bolt launches DoS attacks that do not saturate system resources. Instead, it leverages the information obtained during detection to construct targeted interference signals that affect the performance of the specific application without overloading the host.

Bolt creates a contentious benchmark that puts pressure in resources the victim application is most sensitive to. The benchmark is constructed by combining the microbenchmarks used to measure the resource pressure of the victim. Since these microbenchmarks are tunable, Bolt configures their intensity to a higher point than their measured pressure c_i during detection. For example, if a victim is detected as memcached, and its most critical resources are the L1-i cache (81% pressure) and the LLC (78% pressure), Bolt uses the two microbenchmarks for L1-i and LLC respectively at a higher intensity than what memcached can tolerate.

Impact: Figure 10 shows the performance degradation caused in correctly- (left) and incorrectly-identified applications (right) by Bolt. Performance is shown in terms of execution time for equal amounts of work. On average, correctly-identified applications degrade by 77% in execution time, and up to 2.2x. Degradation is much more pronounced for interactive workloads like key-value stores, with tail latency increasing by 8-140x, a dramatic impact for applications with strict tail latency SLAs. Incorrectly-identified workloads experience moderate performance degradation, as Bolt introduces contention in non-critical shared resources.

We now examine the impact of the DoS attack on utilization. If interference translates to resource saturation, there is a high probability that the DoS will be detected by mitigation mechanisms and the victim migrated to a new machine. The experimental cluster supports live migration. If CPU utilization (sampled every 1 sec) exceeds 70% the victim is migrated to an unloaded host. Figure 11 compares the tail latency and CPU utilization with Bolt to that of a naïve DoS that saturates the CPU through a compute-intensive kernel. We focus on a single victim VM running memcached. The overhead of migration (time between initiating migration and latency returning to normal levels) for this VM is 8 seconds. Performance degradation is similar for both systems until time 80sec, at which point the victim is migrated to a new host. The reason for migration is the compute-intensive kernel causing utilization to exceed 70%. While during migration performance continues to degrade, once the VM resumes in the new server, latency returns to nominal levels. In contrast, Bolt keeps utilization low, and impacts the performance of memcached beyond the 80sec mark.

5.2. Resource Freeing Attack

Attack setting: A resource freeing attack (RFA) has the goal of modifying the workload of a victim VM in a way

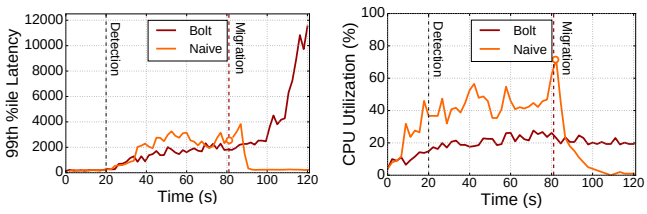


Figure 11: Latency and utilization with Bolt and a naïve DoS attack that saturates CPU resources.

that frees up resources for the adversary, improving its performance [17, 78]. The adversarial VM consists of two components, a *beneficiary* and a *helper*. The beneficiary is the program whose performance the attacker wants to improve, and the helper is the program that forces the victim to yield its resources. The RFA works by adding load in the victim's critical resource, causing other resources to free up. For example, if the victim is a memory-bound Spark job running *k-means*, introducing additional traffic in the memory system will result in Spark stalling in memory, and lessening its pressure in the other resources, until it can reclaim its required memory bandwidth. When launched in a public cloud, the victim of an RFA ends up paying more and achieving worse performance compared to running in isolation.

We now create a proof of concept RFA for a webserver, a network-bound Hadoop job and a memory-bound Spark job. Launching RFAs requires in depth knowledge of the resource requirements of the victim [17]. Bolt resolves this issue by leveraging data mining to determine the dominant resource of each workload. Once determined, the runtime then applies a target helper program that saturates the critical resource. For the webserver, this is a CPU-intensive benchmark launching CGI requests, thus causing the victim to saturate its CPU usage, freeing up cache resources, and servicing fewer real user requests. For the Hadoop job, we use a network-intensive benchmark similar to `iperf`, which saturates network bandwidth, and frees up CPU and memory resources for the beneficiary. Similarly, for Spark we use a streaming memory benchmark that slows down k-means and frees network and compute resources for the adversary. The selection of the beneficiary is of lesser importance; without loss of generality we select `mcf`, a CPU- and memory-intensive benchmark from SPECCPU2006. The same methodology can be used with other beneficiaries, conditioned to their critical resource not overlapping with the victim's.

Impact: Table 3 shows the performance degradation for the three victim applications, and the improvement in execution time for the beneficiary. The webserver suffers the most in terms of queries per second, as the helper's CGI requests pollute its cache, preventing it from servicing legitimate user requests. Hadoop and Spark experience significant degradations in execution time, due to network and memory bandwidth saturation respectively. Execution time for `mcf` improves by 16-28%, benefiting from the victim stalling in its critical

Victim		Beneficiary		Target
App	Perf	App	Perf	Resource
Apache Webserver	-64% (QPS)	`mcf`	+24%	CPU
Hadoop (*SVM*)	-36% (Exec.)	`mcf`	+16%	Network BW
Spark (*k-means*)	-52% (Exec.)	`mcf`	+28%	Memory BW

Table 3: Resource freeing attack impact.

resource to improve its cache and CPU usage.

5.3. VM Co-residency Detection

Attack setting: Sections 3.4 and 4 showed that we can determine the applications sharing a cloud infrastructure. However, a malicious user is rarely interested in any random service running on a public cloud. More often, they target a specific workload, e.g., a competitive e-commerce site. Therefore they need to pinpoint where a target application resides in a practical manner. This requires a *launch strategy* and a mechanism for *co-residency detection* [18]. The attack is practical if the target is located with high confidence, in reasonable time and with modest resource costs. Bolt enables this attack to be carried out in a practical way, and remains resilient against co-residency detection defense mechanisms that cloud providers have put in place, such as virtual private clouds (VPCs) which make internal IP addresses private to a single tenant. Once a target VM is located, the adversary can then launch DoS or RFAs as previously described. Co-residency detection attacks rely on leakage of logical information, e.g., IP address, or on observing the performance impact of a side-channel attack due to resource contention. Bolt relies on a variation of the latter approach.

Assume a system of N servers. A target victim user launches k VMs. The adversary launches n malicious VMs. Instances are launched simultaneously to avoid malicious-malicious instance co-residency [18]. The probability that at least one adversarial VM will be co-scheduled with one victim instance is: $P(f) = 1-(1-k/N)^n$. The adversary then uses a two-process detection scheme. The *sender* is a process that creates contention that degrades the victim's performance. The *receiver* is the process that detects this degradation, running either on the same host as the victim (*cooperative detection*) or externally (*uncooperative detection*). We assume an uncooperative victim, which is the more general case. Bolt works as the sender and uses its detection strategy to determine the type of co-residents on each sampled host and determine if one or more are of the desired type. Assume m VMs of the desired type have been detected in the sample set. Bolt now needs to prune down this space to VM(s) from the specific victim. It launches an external receiver aiming at the victim that uses a public channel, e.g., HTTP, or key-value protocol, to ping the victim service. This is the only point of communication with the victim. At the same time the sender introduces contention in the resources the victim is most sensitive to. If the sender and victim are co-residents the receiver's user requests will be slower due to contention. Bolt quickly prunes down the sampled VMs by determining the type of co-residents. In a large cluster this reduces both the time it takes to pinpoint a victim, and the cost of the adversary's resources. It also increases the confidence in the detection, in the presence of noise from other co-residents.

Impact: We evaluate co-residency detection on the same 40-node cluster. The victim is a SQL server, instantiating a single VM. The cluster also hosts 7 other VMs running SQL. The other running jobs include key-value stores, Hadoop and Spark analytics. Bolt launches 10 senders simultaneously on randomly-selected machines ($P(f) = 0.66$), and detects 3 VMs running SQL in the sample set. It then introduces memory interference, while the receiver launches a large number of SQL queries. While the same set of queries return with 8.16msec mean latency without interference, latency is now measured at 26.14msec. Therefore we conclude that a $\sim 3\times$ latency increase shows that one of the co-residents is the victim SQL server. Detection required 6sec from instantiation to receiver detection, and 11 adversary VMs.

6. Improving Security through Resource Isolation

Since interference is at the heart of Bolt's detection methodology, isolation mechanisms that attenuate resource contention should reduce Bolt's accuracy. We first evaluate to what extent existing isolation techniques mitigate security vulnerabilities, and then highlight trade-offs between security and performance (or utilization). We use the same experimental setting as in the controlled experiment of Section 3.4.

We evaluate 4 resource-specific isolation techniques with 3 settings for OS level isolation mechanisms: *baremetal*, *containerized*, and *virtualized*. The baremetal system does not employ any OS level isolation. For the containerized setup, we use Linux containers (`lxc`), and control the cores for each application via `cpuset cgroups`. Both containers and VMs constrain memory capacity. Baremetal experiments do not enforce memory capacity allocations, and the Linux scheduler is free to float applications across cores.

First, we introduce *thread pinning* to physical cores, to constrain interference from scheduling actions, like context switching. The number of cores an application is allocated can change dynamically, and is limited by how fast Linux can migrate tasks, typically in the tens of milliseconds.

For *network isolation*, we use the outbound network bandwidth partitioning capabilities of Linux's traffic control. Specifically, we use the `qdisc` [79] scheduler with hierarchical token bucket queueing discipline (HTB) to enforce outgress bandwidth limits. The limits are set to the maximum traffic burst rate for each application to avoid contention (`ceil` parameter in HTB interface). Ingress network bandwidth isolation has been extensively studied in previous work [80]; these approaches can be applied here as well.

For *DRAM bandwidth isolation*, there is no commercially

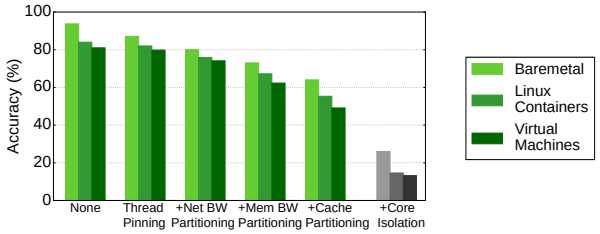


Figure 12: Detection accuracy with isolation techniques.

available partitioning mechanism. To enforce bandwidth isolation we use the following approach: we monitor the DRAM bandwidth usage of each application in software (through performance counters) [29] and modify the scheduler to only colocate applications on machines that can accommodate their aggregate peak memory bandwidth requirements. [6]

Finally, for *last level cache (LLC) isolation*, we use the Cache Allocation Technology (CAT) available in recent Intel chips [81]. CAT partitions the LLC in ways, which in a highly-associative cache enables defining non-overlapping partitions at the granularity of a few percent of the LLC capacity. Each co-resident is allocated one partition configured to its current capacity requirements [82]. Partitions can be resized at runtime by reprogramming specific low-level registers (MSRs); changes take effect after a few milliseconds.

We add one isolation mechanism at a time in the three system settings (baremetal, containers, VMs). Figure 12 shows the impact isolation techniques have on the application detection accuracy of Bolt. The application scenario is the same as in the controlled experiment of Section 3.4.

As expected, when no isolation is used, baremetal allows a significantly higher detection accuracy than container- and VM-based systems; mostly due to the latter constraining core and memory capacity usage. As a result, introducing thread pinning benefits baremetal the most, since it reduces contention in cores. It also benefits container- and VM-based setups to a lesser degree, by eliminating unpredictability introduced by the Linux scheduler (e.g., context switching) [83]. The dominant resource of each application determines which isolation technique benefits it the most. Thread pinning mostly benefits workloads bound by on-chip resources, such as L1, L2 caches and cores. Adding network bandwidth partitioning lowers detection accuracy for all three settings almost equally. It primarily benefits network-bound workloads, for which network interference conveys the most information for detection. Memory bandwidth isolation further reduces accuracy by 10% on average, benefiting jobs dominated by DRAM traffic. Finally, cache partitioning has the most dramatic reduction in accuracy, especially for LLC-bound workloads. We attribute this to the importance cache pressure has as a detection signal. The number of co-residents also affects the extent to which isolation helps. The more co-scheduled applications exist per machine, the more isolation techniques degrade accuracy, as they make distinguishing between co-residents harder.

Unfortunately, even with all previous techniques employed accuracy is still 50%. There are two reasons for this: current techniques are not fine-grain enough to allow strict and scalable isolation, and core resources (L1 i/d, L2 caches, CPU) are still prone to interference due to contending hyperthreads. To evaluate the latter hypothesis, we modify the scheduler, such that hyperthreads of different instances are never scheduled on the same physical core, e.g., if an application needs 7vCPUs it will be allocated 4 dedicated physical cores. The grey bars of Figure 12 show the detection accuracy. Baremetal instances still allow certain applications to be detected, but for containerized and virtualized settings, accuracy drops to 14%, since cores are never shared across applications. The remaining accuracy corresponds to disk bandwidth-bound workloads. Improving security, however, comes at a significant performance penalty of 34% on average in execution time, as threads of the same workload are forced to contend with one another. Alternatively, users can overprovision their resource reservations to avoid degradation, which in turn results in a 45% drop in utilization. This means that the cloud provider cannot leverage CPU idleness to share machines, decreasing the cost benefits of cloud computing. Note that enforcing core isolation alone is also not sufficient, as it allows a detection accuracy of 46%.

Discussion: The previous analysis highlights a design problem with current datacenter platforms. Traditional multicore architectures are prone to contention, which will only worsen as more cores are integrated in each server, and multi-tenancy becomes more pronounced. Existing isolation techniques are insufficient to mitigate security vulnerabilities, and techniques that provide reasonable security guarantees either sacrifice performance or cost efficiency, through low utilization. This highlights the pressing need for new *fine-grain*, and *coordinated* isolation techniques that guarantee security at high utilization for core and uncore resources.

7. Conclusions

We have presented Bolt, a practical system that identifies the type and characteristics of applications running on shared cloud systems, and enables attacks that degrade their performance. Bolt relies on fast and online data mining techniques to project the interference victims introduce in shared resources against information on previously-seen workloads. In a 40-server cluster Bolt correctly identifies 81% out of 108 workloads, and degrades tail latency by up to 140x. We also used Bolt to analyze the most popular services on EC2, across time and regions. Finally, we show that, while existing isolation techniques are helpful, they are not sufficient to mitigate such attacks. Bolt reveals real, easy-to-exploit threats in public clouds. We hope that this work will motivate cloud providers and computer scientists to develop and deploy stricter isolation primitives in cloud servers.

[6] This requires extensive application knowledge, and is used simply to highlight the benefits of DRAM bandwidth isolation.

References

[1] L. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.

[2] "Amazon ec2." http://aws.amazon.com/ec2/.

[3] "Google container engine." https://cloud.google.com/container-engine.

[4] J. Mars and L. Tang, "Whare-map: heterogeneity in "homogeneous" warehouse-scale computers," in *Proceedings of ISCA*, Tel-Aviv, Israel, 2013.

[5] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, USA, 2013.

[6] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *Proceedings of EuroSys*, Paris,France, 2010.

[7] R. B. Lee, "Rethinking computers for cybersecurity," *IEEE Computer*, vol. 48, no. 4, pp. 16–25, 2015.

[8] F. Liu, L. Ren, and H. Bai, "Mitigating cross-vm side channel attack on multiple tenants cloud platform," in *Journal of Computers, Vol 9, No 4 (2014), 1005-1013, April 2014*.

[9] H. Wang, H. Zhou, and C. Wang, "Virtual machine-based intrusion detection system framework in cloud computing environment," in *Journal of Computers, October 2012*.

[10] S. Gupta and P. Kumar, "Vm profile based optimized network attack pattern detection scheme for ddos attacks in cloud," in *Proc. of SSCC*, Mysore, India, 2013.

[11] A. Bakshi and Y. B. Dujodwala, "Securing cloud from ddos attacks using intrusion detection system in virtual machine," in *Proc. of the 2010 Second International Conference on Communication Software and Networks (ICCSN)*, 2010.

[12] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, 2012.

[13] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, 2009.

[14] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2011.

[15] Z. Xu, H. Wang, and Z. Wu, "A measurement study on co-residence threat inside the cloud," in *Proc. of the 24th USENIX Security Symposium (USENIX Security)*, Washington, DC, 2015.

[16] A. Herzberg, H. Shulman, J. Ullrich, and E. Weippl, "Cloudoscopy: Services discovery and topology mapping," in *Proceedings of the ACM Workshop on Cloud Computing Security Workshop (CCSW)*, Berlin, Germany, 2013.

[17] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense)," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, 2012.

[18] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A placement vulnerability study in multi-tenant public clouds," in *Proc. of the 24th USENIX Security Symposium (USENIX Security)*, Washington, DC, 2015.

[19] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, pp. 42–51, Sept. 2007.

[20] D. Mangot, "Ec2 variability: The numbers revealed." http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_re%vealed.

[21] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proceedings VLDB Endow.*, vol. 3, pp. 460–471, Sept. 2010.

[22] C. Delimitrou and C. Kozyrakis, "Optimizing Resource Provisioning in Shared Cloud Systems," *CSTR 2014-06 11/25/2014, November 2014*.

[23] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of ec2 cloud computing services for scientific computing," in *Lecture Notes on Cloud Computing*, Volume 34, p.115-131, 2010.

[24] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *Proceedings of CCGRID*, Newport Beach, CA, 2011.

[25] S. Rehman and M. Sakr, "Initial findings for provisioning variation in cloud computing," in *Proceedings of CloudCom*, Indianapolis, IN, 2010.

[26] Y. E. Khamra, H. Kim, S. Jha, and M. Parashar, "Exploring the performance fluctuations of hpc workloads on clouds," in *Proceedings of CloudCom*, Indianapolis, IN, 2010.

[27] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proc. of the ACM Workshop on Cloud Computing Security (CCSW)*, Chicago, IL, 2009.

[28] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," in *Proceedings of the 38th annual International Symposium in Computer Architecture (ISCA-38)*, San Jose, CA, June, 2011.

[29] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, 2015.

[30] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, 2012.

[31] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Seawall: Performance isolation for cloud datacenter networks," in *Proc. of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, Boston, MA, 2010.

[32] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of l2 cache covert channels in virtualized environments," in *Proc. of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW)*, Chicago, IL, 2011.

[33] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler, "Detecting co-residency with active traffic analysis techniques," in *Proc. of the ACM Workshop on Cloud Computing Security Workshop (CCSW)*, Raleigh, NC,2012.

[34] Y. Han, T. Alpcan, J. Chan, and C. Leckie, "Security games for virtual machine allocation in cloud computing," in *4th International Conference on Decision and Game Theory for Security*, Fort Worth, TX, 2013.

[35] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *ACM SIGCOMM Computer Communication Review (CCR)*, Apr. 2004.

[36] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, "More for your money: Exploiting performance heterogeneity in public clouds," in *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, San Jose, CA, 2012.

[37] J. Huang, D. M. Nicol, and R. H. Campbell, "Denial-of-service threat to hadoop/yarn clusters with multi-tenancy," in *Proc. of the IEEE International Congress on Big Data*, Washington, DC, 2014.

[38] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of network-based defense mechanisms countering the dos and ddos problems," *ACM Comput. Surv.*, vol. 39, Apr. 2007.

[39] M. Darwish, A. Ouda, and L. F. Capretz, "Cloud-based ddos attacks and defenses," in *Proc. of i-Society*, Toronto, ON, 2013.

[40] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th Conference on USENIX Security Symposium*, Washington, DC, 2003.

[41] J. Edge, "Denial of service via hash collisions." http://lwn.net/Articles/474912/, January 2012.

[42] Y. Zhang and M. K. Reiter, "Duppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, 2013.

[43] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. of IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, 2015.

[44] Y. Yarom and K. Falkner, "Flush+reload: a high resolution, low noise, l3 cache side-channel attack," in *Proc. of the 23rd Usenix Security Symposium*, San Diego, CA, 2014.

[45] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ""ooh aah... just a little bit" : A small amount of side channel can go a long way," in *Proc. of the International Cryptographic Hardware and Embedded Systems Workshop (CHES)*, Busan, South Korea, 2014.

[46] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Scottsdale, AZ, 2014.

[47] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *Proc. of the 21st USENIX Conference on Security Symposium (USENIX Security)*, Bellevue, WA, 2012.

[48] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proceedings of the 2013 International Workshop on Security in Cloud Computing, SCC@ASIACCS*, Hangzhou, China, 2013.

[49] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-vm side-channels," in *Proc. of the 23rd Usenix Security Symposium*, San Diego, CA, 2014.

[50] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Portland, OR, 2012.

[51] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, pp. 431–473, May 1996.

[52] C. Delimitrou and C. Kozyrakis, "iBench: Quantifying Interference for Datacenter Workloads," in *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*, Portland, OR, September 2013.

[53] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, USA, 2014.

[54] R. Burke, "Hybrid recommender systems: Survey and experiments," *User Modeling and User-Adapted Interaction*, vol. 12, pp. 331–370, Nov. 2002.

[55] A. Felfernig and R. Burke, "Constraint-based recommender systems: Technologies and research issues," in *Proceedings of the ACM International Conference on Electronic Commerce (ICEC)*, Innsbruck, Austria, 2008.

[56] A. Gunawardana and C. Meek, "A unified approach to building hybrid recommender systems," in *Proc. of the Third ACM Conference on Recommender Systems (RecSys)*, New York, NY, 2009.

[57] J. Zhu, P. He, Z. Zheng, and M. R. Lyu, "Towards online, accurate, and scalable qos prediction for runtime service adaptation," in *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, Madrid, Spain, 2014.

[58] I. H. Witten, E. Frank, and G. Holmes, *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd Edition.

[59] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of the International Conference on Computational Statistics (COMPSTAT)*, Paris, France, 2010.

[60] K. C. Kiwiel, "Convergence and efficiency of subgradient methods for quasiconvex minimization," in *Mathematical Programming (Series A) (Berlin, Heidelberg: Springer) 90 (1): pp. 1-25, 2001.*

[61] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proceedings of the 38th annual international symposium on Computer architecture*, pp. 319–330, 2011.

[62] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, Minneapolis, MN, 2014.

[63] Mahout. http://mahout.apache.org/.

[64] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of NSDI*, San Jose, CA, 2012.

[65] B. Fitzpatrick, "Distributed caching with memcached," in *Linux Journal, Volume 2004, Issue 124, 2004.*

[66] "Apache cassandra." http://cassandra.apache.org/.

[67] E. Brewer, "Kubernetes: The path to cloud native." http://acmsocc.github.io/2015/keynotes/socc15-keynote2.pdf, SOCC Keynote, August 2015.

[68] "Host server cpu utilization in amazon ec2 cloud." http://huanliu.wordpress.com/2012/02/17/host-server-cpu-utilization-in-amazon-ec2-cloud/.

[69] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of NSDI*, Boston, MA, 2011.

[70] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of EuroSys*, Prague, Czech Republic, 2013.

[71] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *Proceedings of the 16th Intl. Conference on Parallel Architecture and Compilation Techniques (PACT)*, Brasov, Romania, 2007.

[72] M. Bhadauria and S. A. McKee, "An approach to resource-aware co-scheduling for cmps," in *Proc. of the 24th ACM International Conference on Supercomputing (ICS)*, Tsukuba, Japan, 2010.

[73] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, 2010.

[74] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram, "Scheduler vulnerabilities and coordinated attacks in cloud computing," *J. Comput. Secur.*, vol. 21, pp. 533–559, July 2013.

[75] "Aws application services." https://aws.amazon.com/products/.

[76] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *Proc. of 16th USENIX Security Symposium on USENIX Security Symposium (SS)*, Boston, MA, 2007.

[77] H. Wang, C. Isci, L. Subramanian, J. Choi, D. Qian, and O. Mutlu, "A-drm: Architecture-aware distributed resource management of virtualized clusters," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE)*, Istanbul, Turkey, 2015.

[78] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Secretly monopolizing the cpu without superuser privileges," in *Proc. of 16th USENIX Security Symposium on USENIX Security Symposium*, Boston, MA, 2007.

[79] M. A. Brown, "Traffic control howto." http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[80] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "Eyeq: Practical network performance isolation at the edge," in *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Lombard, IL, 2013.

[81] "Intel ®64 and IA-32 Architecture Software Developer's Manual, vol3B: System Programming Guide, Part 2, September 2014."

[82] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, 2006.

[83] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of EuroSys*, Amsterdam, The Netherlands, 2014.